Записки за упражнения по ДАА

Тодор Дуков

11 юли 2024 г.

Съдържание

Глава 1

Въведение в алгоритмите и асимптотичния анализ

1.1 Що е то алгоритъм?

Алгоритмите се срещат навсякъде около нас:

- рецептите са алгоритми за готвене
- сутрешното приготвяне
- придвижването от точка A до точка B
- търсенето на книга в библиотеката

Въпреки това е трудно да се даде формална дефиниция на това какво точно е алгоритъм. На ниво интуиция, човек може да си мисли, че това просто е някакъв последователен списък от стъпки/инструкции, които човек/машина трябва да изпълни. Други начини човек да си мисли за алгоритмите, са:

- програми – обикновено така се реализират алгоритми
- машини на Тюринг, крайни (стекови) автомати или формални граматики
- частично рекурсивни функции

Един програмист в ежедневието си постоянно пише алгоритми за да решава различни задачи/проблеми. Една задача може да се решава по много начини, някои по-добри от други. Добрият програмист, освен че ще намери решение на проблема, той ще намери най-доброто решение (или поне достатъчно добро за неговите цели).

1.2 Какво означава добро решение?

Хубаво е човек да се води по следните (неизчерпателни) критерии:

- решението трябва да е коректно – ако алгоритъмът работи само през 50% от времето, най-вероятно можем да се справим по-добре
- решението трябва да е бързо – ако алгоритъмът ще завърши работа след като всички звезди са измрели, то той практически не ни върши работа
- решението трябва да заема малко памет – ако алгоритъмът по време на своята работа се нуждае от повече памет, колкото компютърът може да предостави, за нас този алгоритъм е безполезен
- решението трябва да е просто – това е може би най-маловажният критерии от тези, но въпреки това е хубаво когато човек може, да пише чист и разбираем код, който лесно се разширява

За да можем да сравняваме алгоритми в зависимост от това колко големи ресурси (време и памет) използват, трябва първо да можем да "измерваме" тези ресурси.

1.3 Как мерим времето и паметта?

Когато пишем алгоритми, имаме няколко базови инструкции (за които предварително сме се уговорили), които ще наричаме **атомарни инструкции**. Тяхното извикване ще отнеме една единица време. **Време за изпълнение** ще наричаме броят на извикванията на атомарните инструкции по време на изпълнение на програмата. Също така числата и символите ще бъдат нашите **атомарни типове данни**, и ще заемат една единица памет. **Паметта**, която една програма заема, ще наричаме максималния брой на единици от атомарни типове данни по време на изпълнение, без да броим входните данни. Обикновено времето и паметта зависят от размера на подадените входни данни. Това означава, че можем да си мислим за времето и паметта като функции на размера на входа. Подходът, който ще изберем, е да сравняваме функциите за време/памет на различните алгоритми асимптотично. Интересуваме се не толкова от конкретните стойности, а от поведението им, когато размерът на входа клони към безкрайност.

1.4 Основни дефиниции

Множеството от функции, които ще анализираме, е

$$\mathcal{F} = \{f \mid f : \mathbb{R}^{\geq 0} \to \mathbb{R} \,\&\, (\exists n_0 > 0)(\forall n \geq n_0)(f(n) > 0)\}.$$

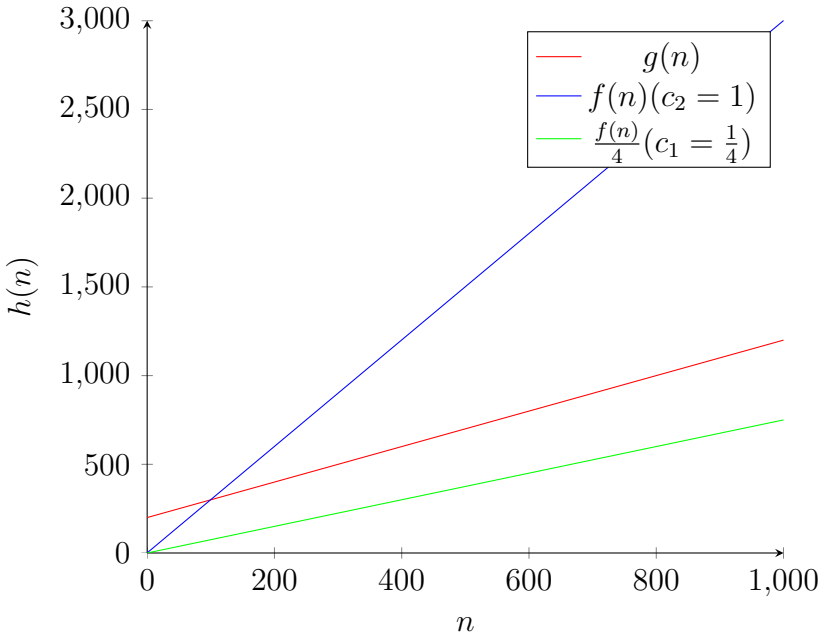
Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\Theta(f) = \{g \in \mathcal{F} \mid (\exists c_1 > 0)(\exists c_2 > 0) \\ (\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n))\}.$$

Може да тълкуваме $\Theta(f)$ като:

"множеството от функциите, които растат със скоростта на f".*

Нека вземем за пример $f(n) = 3n + 1$ и $g(n) = n + 200$:



На картинката се вижда как от един момент нататък, функцията f остава "заключена" между $c_1 \cdot g$ и $c_2 \cdot g$. Точно заради това $g \in \Theta(f)$.

Забележка. Вместо да пишем $g \in \Theta(f)$, ще пишем $g = \Theta(f)$ или $g \asymp f$.

*точност до константен множител и константно събираемо

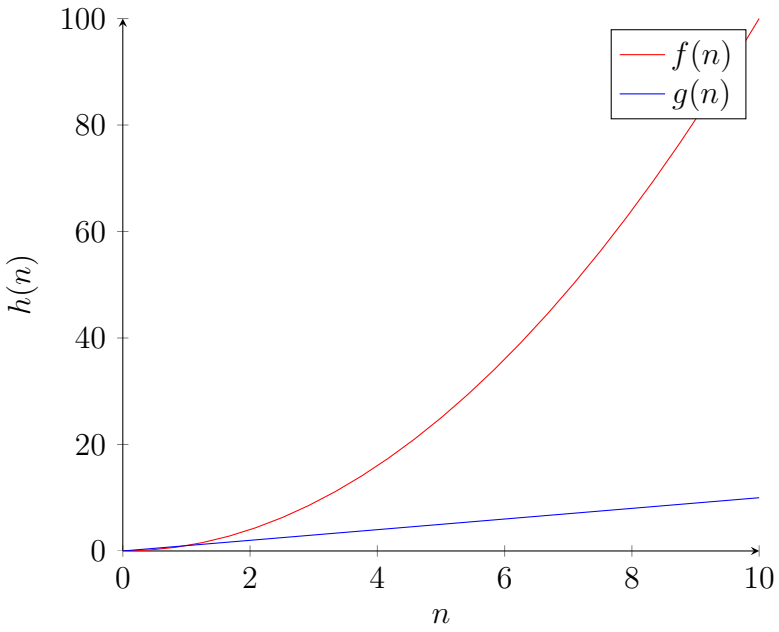
Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$O(f) = \{g \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(g(n) \leq c \cdot f(n))\}.$$

Може да тълкуваме $O(f)$ като:

"множеството от функциите, които не растат по-бързо от f".*

Тук заслабваме условията от $\Theta(f)$ като искаме само горната граница.
За пример човек може да вземе $f(n) = n^2$ и $g(n) = n$:



Забележка. Вместо да пишем $g \in O(f)$, ще пишем $g = O(f)$ или $g \preceq f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$o(f) = \{g \in \mathcal{F} \mid (\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(g(n) < c \cdot f(n))\}.$$

Може да тълкуваме $o(f)$ като:

"множеството от функциите, които растат по-бавно от f".*

Разликата между $O(f)$ и $o(f)$ е строгото неравенство и универсалният квантор в началото. Лесно се вижда, че $o(f) \subseteq O(f)$. Тук изключваме функциите от същия порядък.

Забележка. Вместо да пишем $g \in o(f)$, ще пишем $g = o(f)$ или $g \prec f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\Omega(f) = \{g \in \mathcal{F} \mid (\exists c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c \cdot f(n) \leq g(n))\}.$$

Може да тълкуваме $\Omega(f)$ като:

"множеството от функциите, които не растат по-бавно от f".*

Това е дуалното множество на $O(f)$.

Забележка. Вместо да пишем $g \in \Omega(f)$, ще пишем $g = \Omega(f)$ или $g \succeq f$.

Дефиниция. За всяка функция $f \in \mathcal{F}$ дефинираме:

$$\omega(f) = \{g \in \mathcal{F} \mid (\forall c > 0)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)(c \cdot f(n) < g(n))\}.$$

Може да тълкуваме $\omega(f)$ като:

"множеството от функциите, които растат по-бързо от f".*

Това е дуалното множество на $o(f)$.

Забележка. Вместо да пишем $g \in \omega(f)$, ще пишем $g = \omega(f)$ или $g \succ f$.

Внимание. Не всички функции от $\mathcal{F}$ са сравними по релациите $\prec, \preceq$ или $\asymp$. За пример човек може да вземе функциите $f(n) = n$ и $g(n) = n^{1+\sin(n)}$. Лесно се вижда, че функцията $g(n)$ "плава" между $n^0 = 1$ и n^2 т.е. няма нито как да расте по-бързо, нито как да расте по-бавно.

Въпреки това, тези релации са сравнително хубави.

Твърдение 1.4.1. *Следните свойства са в сила:*

- $\asymp$ *е релация на еквивалентност;*
- $\prec$ *и* $\succ$ *са транзитивни и антирефлексивни;*
- $\preceq$ *и* $\succeq$ *са транзитивни и рефлексивни.*

Доказателството на това твърдение оставяме за упражнение на читателя. То е една елементарна разходка из дефинициите.

1.5 Полезни свойства

Тук ще изброим няколко свойства, които много често се ползват в задачите:

- Нека $f, g \in \mathcal{F}$ и $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = l$ (тук искаме границата да съществува). Тогава:
  — ако $l = 0$, то $f \prec g$
  — ако $l = \infty$, то $f \succ g$
  — в останалите случаи $f \asymp g$

- $f + g \asymp \max\{f, g\}$ за всяко $f, g \in \mathcal{F}$

- $c \cdot f \asymp f$ за всяко $f \in F$ и $c > 0$

- $f \asymp g \iff f^c \asymp g^c$ за всяко $f, g \in F$ и $c > 0$

- $O(f) \cap \Omega(f) = \Theta(f)$ за всяко $f \in \mathcal{F}$

- $o(f) \cap \omega(f) = O(f) \cap \omega(f) = o(f) \cap \Omega(f) = \varnothing$ за всяко $f \in \mathcal{F}$

- $f \prec g \iff g \succ f$ и $f \preceq g \iff g \succeq f$ за всяко $f, g \in \mathcal{F}$

- ако $f \prec g$, то $c^f \prec c^g$ за всяко $f, g \in \mathcal{F}$ и $c > 1$

- ако $\log_c(f) \prec \log_c(g)$, то $f \prec g$ за всяко $f, g \in \mathcal{F}$ и $c > 1$

- ако $c^f \asymp c^g$, то $f \asymp g$ за всяко $f, g \in \mathcal{F}$ и $c > 1$

- ако $f \asymp g$, то $\log_c(f) \asymp \log_c(g)$ за всяко $f, g \in \mathcal{F}$ и $c > 1$

- тъй като $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$, то $\log_a(n) \asymp \log_b(n)$ – вече ще пишем само $\log(n)$ като ще имаме предвид $\log_2(n)$

- $n! \asymp \sqrt{n}\frac{n^n}{e^n}$ - апроксимация на Стирлинг

- $\log(n!) \asymp n \log(n)$

- $\log(n) \prec n^k \prec 2^n \prec n! \prec n^n \prec 2^{n^2}$ за всяко $k \geq 1$

1.6 Задачи

Задача 1.1. Да се сравнят асимптотично следните двойки функции:

1. $f(n) = \log(\log(n))$ и $g(n) = \log(n)$
2. $f(n) = 5n^3$ и $g(n) = n\sqrt{n^9 + n^5}$
3. $f(n) = n5^n$ и $g(n) = n^2 3^n$
4. $f(n) = n^n$ и $g(n) = 3^{n^2}$
5. $f(n) = 3^{n^2}$ и $g(n) = 2^{n^3}$

Задача 1.2. Да се докаже, че $\sum_{i=0}^{n} i^k \asymp n^{k+1}$

Задача 1.3. Да се подредят по асимптотично нарастване следните функции:

$f_1(n) = n^2$ $\quad f_2(n) = \sqrt{n}$ $\quad f_3(n) = \log^2(n)$ $\quad f_4(n) = \sqrt{\log(n)!}$

$f_5(n) = \sum_{k=2}^{\log(n)} \frac{1}{k}$ $\quad f_6(n) = \log(\log(n))$ $\quad f_7(n) = 2^{2^{\sqrt{n}}}$ $\quad f_8(n) = \binom{\binom{n}{3}}{2}$

$f_9(n) = 2^{n^2}$ $\quad f_{10}(n) = 3^{n\sqrt{n}}$ $\quad f_{11}(n) = 2^{\binom{n}{2}}$ $\quad f_{12}(n) = \sum_{k=1}^{n^2} \frac{1}{2^k}$

Глава 2

Анализ на сложността на итеративни алгоритми

2.1 Как анализираме един алгоритъм по сложност?

Нека започнем с един прост пример:

```
1  find(A[1...n] ∈ array(ℤ); v ∈ ℤ):
2      for i ← 1 to n:
3          if A[i] = v: return i
4
5      return −1
```

Да кажем, че искаме да проверим броя на инструкциите, която тази функция ще изпълни, преди да приключи работата си. Точен отговор не може да се даде. В зависимост от това къде се намира v във $A[1 \dots n]$, алгоритъмът може да приключи много бързо или много бавно. Можем да дадем горна и долна граница на бързодействието.
Ако v се намира в началото, то ще сме направили само следните 4 операции:

- да инициализираме променливата i със 0;
- да проверим верността на $i < n$;
- да проверим верността на $A[i] = v$;
- да върнем i т.е. 1.

Нека сега да помислим какво ще стане в най-лошия случай (обикновено от тези ще се интересуваме) – v не участва в $A[1 \dots n]$. Тогава n пъти ще изпълним следните 3 операции:

- проверяваме верността на $i \leq n$;
- проверяваме верността на $A[i] = v$;
- увеличаваме i с 1.

Освен тези $3n$ операции, преди всичко трябва да инициализираме променливата i със 1, да се направи последната проверка на верността на $i \leq n$ (която ще ни изкара от цикъла), и да върнем -1. Общо излизат $3n + 3$ операции.
Така виждаме, че в зависимост от входните данни, алгоритъмът приключва работа за поне 4 стъпки и най-много $3n + 3$ стъпки. Такъв алгоритъм ще казваме, че има сложност по време $O(n)$. Разбира се, няма да е грешно и да кажем, че алгоритъмът има сложност по време $\Omega(1)$, но това не ни дава никаква информация, защото всеки алгоритъм има такава сложност. Също така, понеже не използваме допълнителни променливи, алгоритъмът ни има константна сложност по памет или сложност по памет $\Theta(1)$.
По-общо казано, се интересуваме от асимптотиката на $T(n)$, където $T(n)$ е броят елементарни инструкции, които алгоритъмът извиква по време на своето изпълнение, при вход с размер n в най-лошият случай.
Тук вход с големина n може да означава различни неща. Ако входът е някакъв масив или множество, то под размер ще разбираме броят на елементи. Ако пък входът е число, то под размер можем да разбираме самата стойност на числото или дължината на двоичния запис.

2.2 Предимствата и недостатъците на този вид анализ

Най-голямото предимство на асимптотичния анализ, е неговата простота. Вместо да влачим някакви константни множители и събираеми, имаме колкото се може по-проста формула, която да описва сложността на нашия алгоритъм. Това дали един алгоритъм работи със две или три стъпки по-бързо/бавно не ни интересува особено много. При много голям вход те ще работят практически еднакво. В някакъв смисъл това ни помага да виждаме по-голямата картинка. Един алгоритъм може да бъде по-бърз от друг, но от по-бърз алгоритъм до по-бърз алгоритъм има голяма разлика.

Нека вземем за пример следната таблица:

n	$\lceil \log_2(n) \rceil$	n	n^2	2^n
1	0	1	1	2
10	4	10	100	1024
100	7	100	10000	число със 31 цифри
10000	13	10000	100000000	число със 3011 цифри
1000000	20	1000000	1000000000000	число със 301030 цифри

Алгоритъм със сложността n^2 ще е по-бавен от алгоритъм със сложност n, обаче скока в бързината е много по-малък от този между 2^n и n^2.
Този подход обаче си има своите недостатъци. Нека разгледаме два алгоритъма със сложности по време съответно n и $2^{2^{2^{1024}}}$. Ние ведната ще се втурнем да кажем, че първият алгоритъм е по-лош. Той е с линейна сложност, а вторият алгоритъм има константна сложност. Обаче преди вторият алгоритъм даде отговор, всички звезди ще умрат т.е. няма да доживем да чуем този отговор. Разбира се, от някъде нататък, за много голями входни данни, първият алгоритъм наистина ще работи по-бавно, но ние никога няма да работим с толкова големи данни. Тогава на практика, първият алгоритъм е по-добър, нищо че асимптотично се води по-лош. Нас това няма да ни интересува в курса по ДАА.

2.3 Сложност по време на някои алгоритми

Нека видим сложността на алгоритъма за сортиране по метода на мехурчето:

```
1  Sort(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n − 1:
3          for j ← 1 to n − i − 1:
4              if A[j] > A[j + 1]:
5                  temp ← A[j]
6                  A[j] ← A[j + 1]
7                  A[j + 1] ← temp
```

В най-лошия случай сложността $T(n)$ на функцията `Sort` е следната:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i-1} 1 = \sum_{i=1}^{n-1} (n-i-1) = (n-2)+(n-3)+\cdots+0 = \frac{(n-2)(n-1)}{2} \asymp n^2.$$

По принцип $T(n)$ трябва да е сума от 4, а не от 1, но такъв константен брой операции, дори и приложени неконстантен брой пъти, не влияят на асимптотичното поведение.

Нека сега разгледаме следният алгоритъм за степенуване:

```
1  Exp(x, y ∈ ℕ):
2      res ← 1
3
4      while y > 0:
5          if y ≡ 1 (mod 2):
6              res ← res · x
7
8          y ← y/2
9          x ← x^2
10
11     return res
```

Той се възползва от простата идея, че за да сметнем да кажем 3^8, можем вместо 8 пъти да умножаваме числото 3, да представим 3^8 като $3^4 \cdot 3^4$. Тогава 3^4 можем да сметнем веднъж, и да го умножим със себе си. Пак можем да представим 3^4 като $3^2 \cdot 3^2$ и да пресметнем 3^2 само веднъж и да го умножим със себе си. Така при по-голяма стойност на y си спестяваме много работа. С уговорката, че умножението е атомарна операция, сложността по време $T(n)$ (n е стойността на y) на функцията `Exp` е следната:

$$T(n) = \sum_{\substack{i=n \\ i \leftarrow \frac{i}{2}}}^{1} 1 = \underbrace{1 + \cdots + 1}_{\substack{\text{колкото пъти} \\ \text{можем да} \\ \text{делим целочислено} \\ n \text{ на 2 преди} \\ \text{да получим 0}}} = \underbrace{1 + \cdots + 1}_{\text{около } \log(n) \text{ пъти}} \asymp \log(n).$$

2.4 Задачи

Задача 2.1. Да се определи сложността по време за функцията:

```
1  F_1(n ∈ ℕ):
2      s ← 0
3
4      for i ← 1 to n:
5          for j ← 1; j ≤ i; j ← 2j:
6              s ← s + ij
7
8      return s
```

Задача 2.2. Да се определи сложността по време за функцията:

```
1  F_2(n ∈ ℕ):
2      s ← 0
3
4      for i ← 1; i^2 ≤ n; i ← i + 3:
5          s ← s + F_1(n)
6
7      return s
```

Задача 2.3. Да се определи сложността по време за функцията:

```
1  F_3(n ∈ ℕ):
2      s ← 0
3
4      for i ← 1; i ≤ n^2; inc(i):
5          for j ← 1; j ≤ 2i; inc(j):
6              if j ≡ 0 (mod 2):
7                  s ← s + F_1(n)
8              else:
9                  s ← s + F_2(n)
10
11     return s
```

Глава 3

Коректност на итеративни алгоритми

3.1 Какво имаме предвид под коректност?

За целите на този курс един алгоритъм ще наричаме **коректен**, ако завършва при всякакви входни данни и връща правилен резултат при всякакви входни данни

Забележка. Въпреки че ние ще имаме това разбиране в курса, на практика тези изисквания невинаги са изпълнени:

- разглеждат се алгоритми, които могат и да не завършват за някои входни данни – от теоретична гледна точка са интересни за хората, които се занимават с теорията на изчислимостта;
- разглеждат се алгоритми, които много често (но не винаги) връщат правилния резултат – обикновено това се прави с цел бързодействие.

3.2 Едно *"ново"* понятие – инвариант

Специално за итеративните алгоритми се въвежда ново понятие - **инвариант**. Това са специални твърдения, свързани с цикъла.
В най-общият случай (за алгоритми) се формулират по следния начин:

"При k-тото достигане на ред l (ако има няколко инструкции казваме преди/след коя се намираме) в алгоритъма $\mathfrak{A}$ е изпълнено *някакво твърдение, зависещо от k и променливите, използвани в $\mathfrak{A}$*".

Доказателството на такива твърдения протича с добре познатата индукция. Първо доказваме базата т.е. какво се случва при първото достигане на цикъла. Индуктивното предположение и индуктивната стъпка се обединяват в *"нова"* фаза, наречена **поддръжка**. Довършителните разсъждения, които по принцип се намират след доказването на твърдението чрез индукция, ще наричаме **терминация**. Накрая показваме, че винаги ще излезнем от цикъла (**финитност**). Обикновено това ще го смятаме за очевидно (най-вече за `for`-цикли).

Внимание. Това, за което се използват инвариантите, е да се докаже коректността на ЕДИН цикъл, не на цял алгоритъм. Когато в алгоритъма ни има няколко цикъла, на всеки от тях трябва да съответства по един инвариант.

3.3 Инвариантите в действие

Нека разгледаме следния алгоритъм за степенуване на 2:

```
1  Pow2(n ∈ ℕ):
2      r ← 1
3
4      for i ← 1 to n:
5          r ← 2r
6
7      return r
```

Инвариант 3.3.1. *При всяко достигане на проверката за край на цикъла (на ред 4) е изпълнено, че* $r = 2^{i-1}$.

Доказателство.
База. Наистина при първото достигане имаме, че $i = 1$ и от там $r = 1 = 2^{i-1}$.
Поддръжка. Нека при някое непоследно достигане твърдението е изпълнено. Тогава преди следващото достигане на проверката на r присвояваме $2r$, като знаем, че преди r е бил 2^{i-1}, и след това на i присвояваме $i + 1$. Така е ясно, че при новото достигане на проверката r ще стане $2 \cdot 2^{i_{old}-1} = 2^{i_{old}} = 2^{i-1}$.
Терминация. Ако е изпълнено условието за край на цикъла, то тогава $i = n+1$, откъдето ще върнем $r = 2^{(n+1)-1} = 2^n$.
Финитност. Величината $n - i$ започва с $n - 1$, и намалява с 1, докато не стигне -1, когато ще излезнем от цикъла. Следователно алгоритъмът винаги завършва.
□

3.4 С инвариантите трябва да се внимава

Един от често срещаните капани, в които попадат хората, е да не си формулират инвариантът добре. Много е важно инвариант да дава достатъчна информация за това което наистина се случва в алгоритъма. За целта ще разгледаме един пример:

```
1  SelectionSort(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n − 1:
3          m ← i
4
5          for j ← i + 1 to n:
6              if A[j] < A[m]:
7                  m ← j
8
9          swap(A[i], A[m])
```

На интуитивно ниво е ясно какво прави кода. Намира най-малкия елемент, и го слага на първо място. След това намира втория най-малък елемент, и го слага на второ място, и т.н.
Нещо, което някои биха се пробвали да направят за първия цикъл, е следното:

При всяко достигане на проверката за край на цикъла на ред 3 *подмасивът* $A[1 \dots i-1]$ *е сортиран.*

Проблемът с това твърдение, е че може много лесно да се измисли алгоритъм, за който това твърдение е изпълнено, и изобщо не сортира елементите в масива:

```
1  TrustMeItSorts(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n:
3          A[i] ← i
```

Очевидно този за този алгоритъм горната инвариант е изпълнена, но той е безсмислен. Получаваме сортиран масив, но за сметка на това губим цялата информация, която сме имали за него.
Нещо друго, което е важно да се направи, е първо да се формулира инвариант за вътрешния цикъл, и после за външния, като тънкият момент тук е, че ще ни трябват допускания за първият инвариант. Идеята е, че външния цикъл разчита на вътрешния да си свърши работата, и обратно вътрешния разчита (не винаги) на външния преди това да си е свършил работата.

Нека покажем как трябва да станат инвариантите, като доказателството оставяме за упражнение на читателя. Нека $A^*[1 \dots n]$ е първоначалната стойност на входния масив.

Инвариант 3.4.1 (вътрешен цикъл)**.** *При всяко достигане на проверката за край на цикъла на ред* 5 *имаме, че* m *е индексът на най-малкия елемент в масива* $A[i \dots j-1]$.

Инвариант 3.4.2 (външен цикъл)**.** *При всяко достигане на проверката за край на цикъла на ред* 2 *имаме, че масивът* $A[1 \dots i-1]$ *съдържа сортирани първите* $i-1$ *по големина елементи на* $A^*[1 \dots n]$*, като останалите са в* $A[i \dots n]$.

Обикновено в доказателството на коректност на алгоритми най-трудното е да се формулира инвариантът. Ако човек има добре формулирана инвариант, доказателството е на първо място възможно, а на второ – по-лесно.

3.5 Подход при задачи с вече даден алгоритъм

Нека разгледаме следния алгоритъм:

```
1  foo(a ∈ ℕ):
2      x ← 6
3      y ← 1
4      z ← 0
5
6      for i ← 0 to a − 1:
7          z ← z + y
8          y ← y + x
9          x ← x + 6
10
11     return z
```

Питаме се какво връща той?
Обикновено в такива задачи трябва да се изпробва алгоритъма върху няколко стойности. Можем да забележим, че:

- `foo`(0) връща 0;
- `foo`(1) връща 1;
- `foo`(2) връща 8 и така нататък.

Вече можем да видим какво прави алгоритъмът – `foo`(a) връща a^3. Нека сега докажем това:

Инвариант 3.5.1. *При всяко достигане на проверката за край на цикъла на ред* 5 *е изпълнено, че:*

- $x = 6(1 + i)$;
- $y = 3i^2 + 3i + 1$;
- $z = i^3$.

Доказателство.
База. При първото достигане имаме, че:

- $i = 0$;
- $x = 6 = 6 \cdot 1 = 6(1 + 0) = 6(1 + i)$;
- $y = 1 = 3 \cdot 0^2 + 3 \cdot 0 + 1 = 3i^2 + 3i + 1$;
- $z = 0 = 0^3 = i^3$.

Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава при влизане в тялото на цикъла:

- z ще стане $z + y \overset{(\text{ИП})}{=} i^3 + 3i^2 + 3i + 1 = (\underbrace{i + 1}_{\text{ново } i})^3$;
- y ще стане $y + x \overset{(\text{ИП})}{=} 3i^2 + 3i + 1 + 6 + 6i = 3(\underbrace{i + 1}_{\text{ново } i})^2 + 3(\underbrace{i + 1}_{\text{ново } i}) + 1$;
- x ще стане $x + 6 \overset{(\text{ИП})}{=} 6(1 + i) + 6 = 6(1 + \underbrace{i + 1}_{\text{ново } i})$.

Терминация. В последното достигане на проверката за край на цикъла имаме, че $i = a$, и тогава на ред 12 алгоритъмът ще върне $z = a^3$.
Финитност. (от тук нататък повечето няма да ги пишем) Величината $a - i$ започва с a, и намалява с 1, докато не стигне 0, когато ще излезнем от цикъла. Следователно алгоритъмът винаги завършва. □

Нека разгледаме още един такъв пример:

```
1  bar(n ∈ ℤ): return √(n²)
2
3  foo(x, y ∈ Z): return (x+y+bar(x−y))/2
4
5  quux(A[1...n] ∈ array(ℤ)):
6      a ← A[1]
7
8      for i ← 2 to n:
9          a ← foo(a, A[i])
10
11     return a
```

Искаме да видим какво връща $\mathfrak{quux}(A[1 \dots n])$. Тук най-трудното, което трябва да се направи, е да се определи какво връщат $\mathfrak{bar}$ и $\mathfrak{foo}$:

- $\mathfrak{bar}(n) = \sqrt{n^2} = |n|$;
- $\mathfrak{foo}(x, y) = \frac{x+y+|x-y|}{2} = \max\{x, y\}$:
  - ако $x \geq y$, то $\frac{x+y+|x-y|}{2} = \frac{x+y+x-y}{2} = \frac{2x}{2} = x = \max\{x, y\}$,
  - ако $x < y$, то $\frac{x+y+|x-y|}{2} = \frac{x+y+y-x}{2} = \frac{2y}{2} = y = \max\{x, y\}$.

Като знаем какво правят $\mathfrak{bar}$ и $\mathfrak{foo}$, можем да забележим, че $\mathfrak{quux}(A[1 \dots n])$ дава най-големия елемент на $A[1 \dots n]$:

Инвариант 3.5.2. *При всяко достигане на проверката за край на цикъла на ред* 15 *е изпълнено, че* $a = \max A[1 \dots i-1]$.

Доказателство.
База. Наистина при първото достигане $a = A[1] = \max A[1 \dots i-1]$.
Поддръжка. Нека твърдението е изпълнено за някое непоследно достигане на проверката за край на цикъла. Тогава като влезем в тялото на цикъла, променливата a става:

$$\begin{aligned}\mathfrak{foo}(a, A[i]) \overset{(\text{ИП})}{=} \mathfrak{foo}(\max A[1 \dots i-1], A[i]) &= \max(\max A[1 \dots i-1], A[i]) \\ &= \max A[1 \dots i] = \max A[1 \dots \underbrace{i+1}_{\text{ново } i} - 1]\end{aligned}$$

Терминация. При последното достигане на проверката за край на цикъла на ред 15 променливата i ще бъде $n+1$, откъдето функцията ще върне точно $a = \max A[1 \dots (n+1)-1] = \max A[1 \dots n]$, с което сме готови. □

3.6 Задачи

Задача 3.1. Да се:

- напише алгоритъм, който сумира числата в един масив;
- докаже неговата коректност;
- изследва сложността му по време и памет.

Задача 3.2. Даден е следният алгоритъм:

```
1  𝔄(A[1...n] ∈ array(ℤ)):
2      for i ← 1 to n − 1:
3          for j ← i + 1 to n:
4              if A[i] = A[j]:
5                  return 𝕋
6
7      return 𝔽
```

1. Какво връща той? Отговорът да се обоснове.
2. Каква е неговата сложност по време и памет?

Задача 3.3. Даден е следният алгоритъм:

```
1  𝔉(n ∈ ℕ):
2      if n < 2:
3          return n
4
5      a ← 0
6      b ← 1
7
8      for i ← 1 to n − 1:
9          t ← a
10         a ← b
11         b ← t + b
12
13     return b
```

Да се докаже, че $\mathfrak{F}(n)$ връща n-тото число на Фибоначи.

Задача 3.4. Даден е следният: алгоритъм:

```
1  Mult(A, B, C ∈ (ℤ)_{n×n})
2      for i ← 1 to n:
3          for j ← 1 to n:
4              s ← 0
5
6              for k ← 1 to n:
7                  s ← s + A[i][k] · B[k][j]
8
9              C[i][j] ← s
```

Да се докаже, че при вход $n \times n$ целочислени матрици A, B и C, функцията `Mult`(A, B, C) записва в C произведението на A и B. Да се намери сложността му по време и памет.

Задача 3.5. Даден е следният алгоритъм:

```
1  NumSlopes(A[1...n] ∈ array(ℤ)):
2      s ← 1
3
4      for i ← 2 to n:
5          if A[i − 1] > A[i]:
6              s ← s + 1
7
8      return s
```

Какво връща `NumSlopes`$(A[1 \ldots n])$? Отговорът да се обоснове.

Задача 3.6. Даден е следният алгоритъм:

```
1  Kadane(A[1...n] ∈ array(ℤ)):
2      max_so_far ← A[1]
3      max_ending_here ← A[1]
4
5      for i ← 2 to n:
6          max_ending_here = max(max_ending_here + A[i], A[i])
7          max_so_far = max(max_ending_here, max_so_far)
8
9      return max_so_far
```

Какво връща `Kadane`$(A[1 \ldots n])$? Отговорът да се обоснове.

Задача 3.7. Даден е следният алгоритъм:

```
1  FindMajority(A[1...n] ∈ array(ℤ)):
2      m ← A[1]
3      c ← 1
4
5      for i ← 2 to n:
6          if c = 0:
7              m ← A[i]
8              c ← 1
9          else if A[i] = m:
10             c ← c + 1
11         else:
12             c ← c − 1
13
14     return m
```

Да се докаже, че при подаден целочислен масив $A[1 \dots n]$, в който има елемент с повече от $\lfloor \frac{n}{2} \rfloor$ срещания, функцията `FindMajority`$(A[1 \dots n])$ ще върне точно този елемент.